2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

Specializing Compiler Optimizations Through Programmable Composition For Dense Matrix Computations

Qing Yi
Dept of Computer Science
University of Colorado,
Colorado Springs, USA
qyi@uccs.edu

Qian Wang
Univ of Colorado, Colorado Springs
Univ of Chinese Academy of Sciences
& Institute of Software, CAS
& SKL of Computer Science, CAS
traz0824@gmail.com

Huimin Cui
SKL of Computer Architecture
Institute of Computing Tech, CAS
Beijing, China
huimin.cui@gmail.com

***Abstract*—General purpose compilers aim to extract the best average performance for all possible user applications. Due to the lack of specializations for different types of computations, compiler attained performance often lags behind those of the manually optimized libraries. In this paper, we demonstrate a new approach, programmable composition, to enable the specialization of compiler optimizations without compromising their generality. Our approach uses a single pass of source-level analysis to recognize a common pattern among dense matrix computations. It then tags the recognized patterns to trigger a sequence of general-purpose compiler optimizations specially composed for them. We show that by allowing different optimizations to adequately communicate with each other through a set of coordination handles and dynamic tags inserted inside the optimized code, we can specialize the composition of general-purpose compiler optimizations to attain a level of performance comparable to those of manually written assembly code by experts, thereby allowing selected computations in applications to benefit from similar levels of optimizations as those manually applied by experts.**

***Keywords*-Computers and information processing; Computer science; Programming; Automatic programming**

I. Introduction

Dense matrix computations, represented by the Basic Linear Algebra Subprograms (BLAS) library, are widely considered a fundamental component of numerical applications. Many compiler optimizations, e.g., those in Fig 2, have been shown to be highly effective for computations similar to the $gemm$ kernel in Fig 1. However, the performance of the compiler-optimized code is often suboptimal when compared to those attained by manually optimized libraries, e.g., MKL [17], ACML [4], and ATLAS [29], which have been supplied by CPU vendors or HPC researchers, often with selected kernels directly implemented in assembly [8]. Developing highly optimized libraries manually, however, is excessively labor intensive and error prone. As the result, not all kernels receive the same level of optimization, and general-purpose applications cannot benefit from the optimizations unless rewritten to invoke the libraries.

A key dilemma faced by general-purpose compilers is that they must use a common set of strategies to attain the best

```
void gemm(int M,int N,int K,double alpha,double *A,int lda,
      double *B, int ldb,double beta,double *C, int ldc)
{     int i, j, k;
  loop1: for (j = 0; j < N; j += 1)
  loop2:   for (i = 0; i < M; i += 1) {
             C[j * ldc + i] = (beta * C[(j * ldc) + i]);
  loop3:     for (k = 0; k < K; k += 1)
               C[j*ldc+i]+=alpha*A[k*lda+i]*B[j*ldb+k];}}
}
```

Figure 1. Example: the matrix-matrix multiplication kernel

```
(1) Loop parallelization: outermost loop (loop1)
(2) Loop blocking: the entire loop nest (loop1 - loop3)
(3) Loop unroll&Jam: unroll outer loops(loop1/loop2), then
      jam unrolled iterations inside the innermost loop3
(4) Array copy: arrays non-contiguously accessed:
     alpha*A[i*lda+k] using blocked layout for loops i,k
     B[j*ldb+k] using blocked layout for loops j,k
(5) Scalar replacement: array references inside loops:
     A[k*lda+i] and B[j*ldb+k] inside loop3;
     C[j*ldc+i] inside loop2
(6) Loop unrolling: innermost loop (loop3)
(7) Strength reduction: array address calculations:
     A[k*lda+i], B[j*ldb+k], and C[j*ldc+i]
(8) SIMD vectorization: innermost loop (loop3)
(9) Loop splitting: remove conditionals inside loops
(10) Peephole opt.: pattern-based assembly-level opt.
```

Figure 2. Example: optimizating the gemm kernel in Fig 1

average performance for all possible user applications. Their optimizations are generally organized as a sequence of independent passes, e.g. polyhedral loop optimization pass [9] vs. register allocation and SIMD vectorization passes [15], [26], [23]. To ensure correctness and profitability, each pass needs to re-analyze the output of previous passes to determine the feasibility of additional optimizations, so that they could refrain from optimizations that are potentially unsafe or non-profitable for some unexpected cases. Since each pass can generate hard-to-analyze code, with redundancies to be removed via later passes [13], much information can be lost from one pass to another, resulting in poor coordination among the optimizations and missed opportunities.

To elaborate, consider the $gemm$ kernel in Fig 1, with a list of optimizations customized for this code shown in Fig 2. Rather than analyzing the source-level representation of the input code to collectively determine the entire collection of applicable optimizations, most existing compilers would use the input code in Fig 1 to directly trigger only the first three loop optimizations in Fig 2, as they all rely on loop

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.14

dependence analysis [2] and can be applied together in a single pass, e.g., via the polyhedral framework [9]. Since array copy must be applied in sync with loop blocking, it is commonly omitted by compilers. The rest of optimizations ((5)-(10)) are typically applied one after another, each as an independent pass, based on analyzing the output of earlier optimizations. The optimization passes may be ordered differently in different compilers, but each compiler typically uses the same ordering for all applications. Since different optimizations often have conflicting objectives, e.g., maximizing cache locality vs. instruction-level parallelism, the lack of coordination among optimizations make their overall impact unpredictable.

Due to the unpredictable interferences among different compiler optimizations, the ordering of optimization phases [19] is a well-known NP-complete problem within compilers, and no single phase ordering is expected to work well for all applications. Instead of attempting to solve this problem in general, we demonstrate a new approach, *programmable composition*, which overcomes this issue through two key technologies: (1) enabling fine-grained coordination among compiler optimizations to minimize their interferences, and (2) specializing the composition of compiler optimizations for known computational patterns so that the highest performance can be attained for these patterns without compromising the average performance attainable for other computations.

As illustrated in Fig 3, we use a single pattern-based analysis phase to recognize a class of dense matrix kernels that benefit from a common set of optimization configurations. The input source code is then annotated with the recognized pattern, shown above $loop1$ in Fig 4(a), and used to trigger a *collective customization* of all the relevant optimizations. The optimizations are then applied in a specialized order customized for the pattern, with each optimization carefully coordinating with others through a set of pre-designated *coordination handles* and dynamically inserted *symbolic tags*. This new optimization approach addresses phase-ordering sensitivity within conventional compilers by enabling all optimizations to be collectively customized based on the source level representation (e.g., AST) of the original input code. No information is lost from re-analyzing optimized code, so no resulting missed opportunities. All optimizations are well coordinated in the process and thus are insensitive to minor variations induced by other optimizations or from the original input code. e.g, slightly different nesting order of the loops or indexing expressions of the arrays in Fig 1. By recognizing the fundamental properties of computations early, our approach is able to address potential safety concerns, e.g., whether the values of $lda/ldb/ldc$ are greater than those of $N/M/K$ in Fig 1, by inserting runtime checks surrounding the entire optimized code without compromising the effectiveness of optimization.

Our contributions include the following.

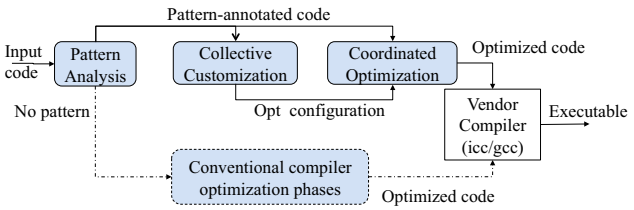


Figure 3. Optimization workflow

- We present a novel approach, *programmable composition*, to support specialized customization and fine-grained coordination of complex compiler optimizations for an important class of dense matrix kernels.
- We applied our prototype compiler to optimize 15 kernels from the BLAS library and 9 application benchmarks from SPLASH-2 [30]. Our optimizer is able to attain a level of performance similar to that attained via manual assembly programming for the kernels and allows such level of optimizations to be attained for selected loops in large applications by automatically recognizing the relevant computation patterns.

We chose dense matrix computations because the optimizations required for these kernels are well known and already integrated as part of most general-purpose compilers, including the POET optimization library [32] which we used to implement our compiler. The same approach, however, can be used to specialize compilers for other important kernels, e.g., stencils and digital signal processing, by similarly recognizing the computation patterns of these kernels, integrating additional required optimizations, and then coordinating the optimizations to maximize their collective effectiveness. Since the conventional optimization passes are used for computations that are not tagged with a recognized pattern, the generality of the compiler is not compromised.

II. Coordinating Compiler Optimizations

A key focus of this paper is to enable fine-grained coordination among independent compiler optimizations to reduce their unexpected interferences. The following introduces two mechanisms, *coordination handles* and *symbolic tagging*, to accomplish this purpose.

A. Coordination Handles

The purpose of each *coordination handle* is to serve as a pre-configured communication channel among different optimizations. Each channel is associated with a single designated piece of code in the original input and is mapped to a permanent name, e.g., the names _1,...,_12 in Fig 4(a). When each designated code piece is modified by some compiler optimization, the modification needs to be coordinated with other optimizations, and the coordination is realized by modifying the corresponding communication channel (i.e., *the coordination handle*) with an equivalent implementation of the original code fragment. Through a set of coordination

handles, each compiler optimization can keep track of modifications to the input code by other optimizations, thereby better accommodating their interferences.

To support effective communication, all the *coordination handles* must be set up before applying any optimization. First, our framework analyzes the internal Abstract Syntax Tree (AST) representation of the original input code to collectively determine all the possible optimizations, illustrated in Fig 2. Then, a coordination handle is created for each piece of the original input code that has been used to configure any of the pre-selected optimizations. The names of these coordination handles are then used in place of the original code pieces to configure the optimizations, shown in Table I. As independently implemented compiler optimizations are applied one after another, each optimization consistently maintains all the relevant coordination handles to ensure the original optimization configurations remain correct irrespective of how many optimizations have already been applied in the process,

Fig 4(a) illustrates the structure of the *gemm* kernel in Fig 1 after a set of *coordination handles* (_1,...,_12) have been created to accommodate interactions among the customized optimizations in Fig 2. Each handle is assigned a name (_1,...,_12) and is associated with a designated fragment of the input code that has been used to configure the optimizations. Some coordination handles, e.g., _1,_2,_7,_8, and _9, are associated with the same fragment of code, e.g., the outermost j loop, because this code piece may be later split into multiple pieces by some optimizations, with each new piece associated with a different handle and serving a different purpose. Fig 4(b) illustrates a number of such situations. For example, after applying OpenMP parallelization and loop blocking, the _8 handle becomes associated with the parallelized j_bk_1 loop, the _9 handle associated with the entire transformed code, the _1 handle associated with the inner blocked loop, while the remaining handles (_7 and _2) stay with the j_bk_2 loop. A new handle, _13, is inserted into the optimized code by the OpenMP parallelization, to collect all the thread-private variables that may be created by the later optimizations.

B. Tracing Optimization-induced Relations

While *coordination handles* can be used to keep track of modifications to the input code and thereby maintain the validity of pre-customized optimization configurations, they must be set up before any optimization is applied. As a result they cannot be used to keep track of the dynamic relations among the modified code fragments by the various optimizations. As illustrated in Fig 4(b), as an input code goes through various optimizations by the compiler, it's internal structure may be dramatically changed, and each piece of code fragment, e.g., a loop or an expression, may be moved around, modified to reflect changes of the surrounding environment, or split into multiple pieces. Such dynamic

```
void gemm(int M, int N, int K, double alpha, double *A, int lda,
          double *B, int ldb, double beta, double *C, int ldc)
{ int i, j, k;
  if (lda>=K && ldb>=K && ldc>=M && no-overlap(A,B,C)) {
      /*@;BEGIN(MM_pattern[precompute=("A","alpha"*"A")])@*/
loop1: _9={_8={_7={_2={_1={for (j = 0; j < N; j += 1)
loop2:   _4={_3={for (i = 0; i < M; i += 1) {
           _10={C[j * ldc + i]} = (beta * C[j * ldc + i]);
loop3:     _6={_5={for (k = 0; k < K; k += 1)
             C[j*ldc+i]+=alpha*{A[k*lda+i]}-11*{B[j*ldb+k]}-12; }}}}}}}}}}}
  else {.....} }
```

(a) After initial optimization customization

```
1: if (lda>=K && ldb>=K && ldc>=M && no-overlap(A,B,C)) {
2:     int j_bk_1,j_bk_2,i_bk_3,k_bk_4;
3:     _9={omp_set_num_threads(2);
4:     #pragma omp parallel for private(_13={k,i,j,j_bk_1,j_bk_2,i_bk_3,k_bk_4})
5:     _8={for (j_bk_1=0; j_bk_1<N; j_bk_1+=256)ℓ1
6:       _7={_2={for (j_bk_2 = 0; j_bk_2 <min(N-j_bk_1); j_bk_2 += 64)tile(ℓ1)
7:         _4={for (i_bk_3 = 0; i_bk_3 < M; i_bk_3 += 64)ℓ2
8:         _6={for (k_bk_4 = 0; k_bk_4 < K; k_bk_4 += 64)ℓ3
9:           _1={for (j = 0; j <min(64,N-j_bk_1-j_bk_2); j += 2)tile(tile(ℓ1))
10:          _3={for (i = 0; i < min(64, M-i_bk_3); i += 1)tile(ℓ2)
11:          _5={for (k = 0; k < min(64, K-k_bk_4); k += 1)tile(ℓ3) {
12:            if (k + k_bk_4 == 0)split_at_begin(tile(ℓ3),1)
13:              _10={C[(j_bk_1+j_bk_2+j)*ldc+(i_bk_3+i)]}
                                  = beta * C[(j_bk_1+j_bk_2+j)*ldc+(i_bk_3+i)];
14:          C[(j_bk_1+j_bk_2+j)*ldc+(i_bk_3+i)]+=alpha*{A[(k_bk_4+k)*lda+(i_bk_3
               +i)]}-11*{B[(j_bk_1+j_bk_2+j)*ldb+(k_bk_4+k)]}-12;
15:          if (j + 1 < min(64,N-j_bk_1-j_bk_2))unroll_check(tile(tile(ℓ1))) {
16             if (k + k_bk_4 == 0)split_at_begin(tile(ℓ3),1)
17:              C[(j_bk_1+j_bk_2+j+1)*ldc+(i_bk_3+i)]
                                  = beta * C[(j_bk_1+j_bk_2+j+1)*ldc+(i_bk_3+i)];
18:            C[(j_bk_1+j_bk_2+j+1)*ldc+(i_bk_3+i)]+=alpha*A[(k_bk_4+k)*lda+(
                 i_bk_3+i)] * B[(j_bk_1+j_bk_2+j+1)*ldb+(k_bk_4+k)];
             } }}}}}}}}}}
  else {.....} }
```

(b) After applying loop parallelization, blocking, and unroll&jam

Figure 4. Coordination handles for the kernel in Fig 1

relations, however, are critical to keeping some of the earlier source-level optimizations, e.g., loop blocking and unrolling, coordinated with later dynamic backend optimizations, e.g., loop splitting and peephole optimizations. For example, without knowing the relations of the various conditional statements introduced by loop blocking and unroll&jam at lines 12, 15, and 16 of Fig 4(b), it is difficult for a later optimization to determine how to split the loops to remove these conditionals based solely on symbolic analysis of the conditionals and the surrounding loop bounds.

We have designed a *symbolic tagging* mechanism to overcome this difficulty, where the internal representations of the various code fragments dynamically generated by optimizations are tagged with unique names and symbolic relations. For example, the blocked loop at line 5 of Fig 4(b) is tagged with a unique name $\ell 1$, and the loop that enumerates its inner tile at line 6 is tagged with $tile(\ell 1)$, a symbolic function indicating its relation with $\ell 1$. The outer and inner tiles of the other blocked loops ($\ell 2$ and $\ell 3$) are tagged similarly to indicate their relations. Further, each of the conditional statements at lines 12, 15 and 16 are tagged with a symbolic relation, e.g., *split_at_begin* and *unroll_check*, to indicate which loops need to be split to eliminate these conditionals. These relational tags are

```
pattern_recognition(input: function to optimize)
    foreach loop nest ℓ in input do
      if (check_loop_shape(ℓ) && check_array_ref_projection(ℓ) &&
          check_parallelizability(ℓ) && check_permutability(ℓ) && check_scalars(ℓ))
      then pattern_annotation(ℓ); endif
    enddo

pattern_annotation(pat: recognized pattern)
    foreach array reference r = arr[sub] in pat do
(1)   bounds = get_dimension_loop_bounds(r, pat);
      if (is_rectangular(bounds)) then matrix_type[arr] ∪ ={rectangular};
      elseif (is_triangular(bounds)) then matrix_type[arr]∪ ={triangular};
      else matrix_type[arr]=unknown; endif
      if size_of (matrix_type[arr]) > 1 then matrix_type[arr] = unknown; endif
(2)   if is_loop_invariant(arr,pat) then
          e = parent_operation(r);
          if is_loop_invariant(e,pat) then precompute[arr]∪={e}; endif
          if size_of(precompute[arr]) > 1 then precompute[arr]=unknown; endif
      endif
(3)   if uses_linearized_subscript(r)
          then runtime_check ∪ = gen_runtime_check(r, bounds); endif
    enddo
(4) if runtime_check ≠ ∅ then pat = insert_runtime_check(pat, runtime_check); endif
    insert_MM_pattern_annotation(pat, matrix_type, precompute);
```

Figure 5. Algorithm: pattern-based optimization analysis

generated independently by each optimization as the various fragments of code are generated. Once inserted, the tags are persistent and immune to modifications to their target unless they are explicitly removed. Consequently, they serve as a dynamic communication mechanism that allows optimizations to coordinate with each other on the fly, e.g., by tagging code fragments to be examined by later optimizations.

III. Specialized Optimization Framework

Fig 3 illustrates the overall workflow of our framework, where the conventional organization of compiler optimizations is augmented with an alternative pattern-driven path, which uses a single *pattern analysis* component to discover computations that have been associated with specialized optimizations. The recognized patterns are then tagged with appropriate annotations, e.g., the *MM_pattern* annotation in Fig 4(a), which are then used by the *collective customization* component to specialize optimizations known to be important for the recognized kernels. Finally, the *coordinated optimization* component systematically invokes the actual optimization implementations with their pre-customized configurations to ensure both correctness and effectiveness. The following details our design of the pattern analysis, collective customization, and the coordinated optimization components for a class of dense matrix computations. Section III-D then discusses the correctness guarantee and the generality of our optimization approach.

A. Pattern Analysis

Our definition of the *MM_pattern* aims to automatically categorize all computations that require the same set of optimizations with similar configurations as those summarized in Fig 2 to attain a highest level of performance. Consequently, our *MM_pattern* analysis algorithm in Fig 5 imposes the following constraints on the input code to ensure both the safety and profitability of the optimizations.

1) Loop shape: the pattern has a single loop nest, with each loop enumerated using a single index variable, and all loops nested inside one another. Loops do not have to be perfectly nested, as astray statements can be embedded inside conditionals after blocking, illustrated at lines 12/16 in Fig 4(b). To ensure safety of optimization, we require that the loop bodies cannot contain any unknown function calls or dynamic jumps.
2) Determinism of array references: the array dimensions accessed by the loops must be an orthogonal projection of the loop iteration space. Specifically, the subscript of each array access must be a linear combination of the loop index variables multiplied by a loop-invariant stride. For example, the reference $A[k*lda+i]$ in Fig 1 is a linear combination of two loop indices, k and i, with accessing strides lda and 1 respectively.
3) Loop parallelizability: the outmost and innermost loops cannot carry any cross-iteration dependences except those that can be categorized as reduction dependences [2] carried by the innermost loop. This requirement ensures the safety of the OpenMP and SIMD vectorization optimizations in Fig 2.
4) Loop permutability: all loops are fully permutable; that is, any dependence carried by a loop is from an earlier iteration to a later iteration. This requirement ensures the safety of loop blocking and unroll&jam [2].
5) Manageability of variables: all memory references in the loops must belong to scalar variables or array references of the following categories: 1) never modified; 2) a distinct element is modified at each iteration of a loop ℓ, and the modification live range is contained inside the single iteration; and 3) reduction variables used to accumulate values from different iterations of an inner loop. This requirement is necessary to ensure the correctness of SIMD vectorization and scalar replacement, by enforcing that all AVX/SSE registers and scalar variables used to replace array references will be properly initialized before used.

Our algorithm in Fig 5 essentially checks each of the above constraints for each code fragment ℓ in the input and inserts annotations to tag ℓ as *MM_pattern* only if all constraints are satisfied. Since each array dimension is required to be an orthogonal projection of its surrounding loops, the dependence constraints can be determined in a much simplified fashion. Our pattern analyzer does not perform pointer aliasing analysis but ensures none of the arrays in the input can be aliased by inserting necessary runtime checks. The identified kernels are tagged with the *MM_pattern* annotation, illustrated in Fig 4(a), with additional specifications (e.g., *precompute*) and runtime safety checks if necessary by determining the following three additional attributes of the pattern.

- *Matrix type:* specifically whether each array in the

pattern represents a rectangular, a triangular, or an unknown matrix type. The evaluation is defined by Step (1) of the *pattern_annotation* function in Fig 5. Here since each array dimension is an orthogonal projection of its surrounding loop iterations, we categorize the shape of each array reference as that of its iteration space, specifically the bounds of its surrounding loops that have nonzero projections. For example, the shape of $A[k * lda + i]$ in Fig 4(a) is the rectangular space $\{$ (k,i) $\mid 0 \leq$ k $<$ K, $0 \leq$ i $<$ M $\}$. Note that if multiple references of an array entail different matrix types, the resulting attribute is set to *unknown*.

- *Precompute:* specifically whether any loop invariant evaluation involving an array reference can be precomputed when the array is copied outside of the loops. Evaluated by Step (2) of the *pattern_annotation* function, this attribute is considered only for read-only arrays and only when the same expression is evaluated whenever the array is referenced. In Fig 4(a), $alpha * A[k * lda + i]$ is an example of such expressions and can be precomputed when array A is copied.
- *Runtime check:* specifically a set of boolean expressions to be checked at runtime to ensure none of the array references in the pattern may carry unexpected dependences. These expressions are collected at Step (3) of the *pattern_annotation* function and used at Step (4) to ensure all optimizations are indeed safe for the annotated *MM_pattern*.

B. Collective Customization Of Optimizations

After recognizing computations of our targeted *MM_pattern* from an input application, our next step examines each annotated code fragment to collectively customize the relevant optimizations. The key strategy is to collect up front all the information required to correctly apply these optimizations to the original code and then ensure all the information stays up-to-date by wrapping up critical pieces of code that have been used to configure the optimizations inside a set of *coordination handles*, illustrated in Fig 4(a). The following details these steps.

Fig 6 summarizes our algorithm for collecting the following pieces of information required to customize our collection of loop and array optimizations.

- The loops targeted by the blocking, unroll&jam, unrolling, and later cleanup optimizations, saved in three *CONFIG* variables, *outer_loops*, *inner_loops*, and *cleanup_scope*, respectively, at lines 8-10 of Fig 6.
- The loop to parallelize via OpenMP, the expected location of the parallelized code, and the thread-private variables inside the to-be-parallelized loop, saved in three *CONFIG* variables, *parallel_loop*, *parallel_top*, and *private_vars*, at lines 11-13 and 31 of Fig 6.
- The blocking factors to be used in loop parallelization, blocking, and unrolling, saved in three *CONFIG* variables, *par_size*, *tile_size*, and *unroll_size*, respectively, at lines 14-16 in Fig 6. These parameters are currently empirically determined by examining the performance of differently optimized code.
- The subscripted references of each array, saved in *CONFIG.arr_refs* at line 22, and for each reference, the loops in *CONFIG.outer_loops* and *CONFIG.inner_loops* that enumerate different array dimensions, saved in *CONFIG.dims_out* and *CONFIG.dims_in* at lines 23-24 of Fig 6.
- Information about each array reference or scalar variable, including whether it is modified/read, needs to be copied, and allows SIMD vectorization (if yes, how to vectorize it), saved in two *CONFIG* variables, *ref_info* and *scalar_info*, at lines 25 and 30 of Fig 6.
- Arrays to be copied, and for each of them, the reference to be optimized for better reuse or the expression to be precomputed during the copy optimization. Saved in *CONFIG.arr_copy* at lines 26 and 28 of Fig 6.

```
setup_loop_opt(pat: a code fragment that belongs to MM_pattern)
1:  outer_loops = ∅; inner_loops=∅; private_vars=∅;
2:  foreach loop ℓ ∈ pat do
3:      t1 = insert_handle(ℓ);
4:      inner_loops = append_to_list(inner_loops, t1);
5:      t2 = insert_handle(t1);
6:      outer_loops = append_to_list(outer_loops, t2);
7:      private_vars=append_to_list(private_vars, loop_index_var(ℓ));
    enddo
8:  CONFIG.outer_loops=outer_loops;
9:  CONFIG.inner_loops=inner_loops;
10: CONFIG.cleanup_scope= insert_handle(first_entry(outer_loops));
11: CONFIG.parallel_loop = insert_handle(CONFIG.cleanup_scope);
12: CONFIG.parallel_top=insert_handle(CONFIG.parallel_loop);
13: CONFIG.private_vars = insert_handle(private_vars);
14: CONFIG.par_size = determine_par_block_factor(CONFIG.parallel_loop,pat);
15: CONFIG.tile_size = determine_blocking_factors(CONFIG.outer_loops,pat);
16: CONFIG.unroll_size = determine_unrolling_factors(CONFIG.inner_loops,pat);

setup_array_opt(pat: a code fragment that belongs to MM_pattern)
17: CONFIG.arr_refs=∅;
18: foreach unique array reference r = arr[sub] in pat:
19:     (ivars, coeffs) = find_loop_index_variables(sub)
20:     loops = map_index_vars_to_loops(ivars);
21:     h = insert_handle(r);
22:     CONFIG.arr_refs = CONFIG.arr_refs ∪ { h };
23:     CONFIG.dims_out[h] = {find_loop_handles(loops, CONFIG.outer_loops)};
24:     CONFIG.dims_in[h] = {find_loop_handles(loops, CONFIG.inner_loops)};
25:     CONFIG.ref_info[h] = uses = categorize_uses(r,pat);
26:     if copy_array(uses) then CONFIG.arr_copy[arr]=h; endif
    enddo
27: foreach (arr, exp) ∈ precompute_annot(pat) do
28:     CONFIG.arr_copy[arr]= insert_handle(exp);
    enddo
29: foreach non-loop-index scalar variable v ∈ pat :
30:     CONFIG.scalar_info[v] = uses = categorize_uses(v,pat);
31:     if is_private(uses) then CONFIG.private_vars ∪ = { v } endif
    enddo
```

Figure 6. Algorithm: collectively customizing optimizations

The *insert_handle* function is invoked at lines 3, 5, 10-13, 21, and 28 of Fig 6 to wrap the various designated pieces of the input code inside coordination handles, illustrated in Fig 4(a), before the coordination handles are saved as values of the configuration variables, so that the content of these configuration variables will to be automatically adjusted as the input code goes through the various optimizations.

Table I illustrates our customized ordering of the

optimization	configuration interface	CONFIG for gemm
OMP parallel	which loop to parallelize thread-private variables parallel block size	parallel_loop=_8 private_vars=_13 par_size=256
blocking	which loop nest to block blocking factors	outer_loops={_2,_4,_6} tile_size={64,64,64}
array copy	which array references to copy loops projected in each reference is each reference read/modified	arr_copy[A]=_11, arr_copy[B]=_12 dims_out[_11]=(_4,_6), dims_out[_12]=(_2,_6) ref_info[_11]=(R,copy,vec), ref_info[_12]=(R,copy,vec)
unroll&Jam	loops to unroll & jam unroll factors	inner_loops={_1,_3,_5} unroll_size={4,1,4}
scalar repl	which array references to replace loops enumerating each reference is each reference read/modified	arr_refs={_10,_11,_12} dims_in[_10]=(_1,_3), dims_in[_11]=(_3,_5), dims_in[_12]=(_1,_5) ref_info[_10]=(RW,reduce), ref_info[_11]=(R,copy,vec), ref_info[_12]=(R,copy,vec)
SIMD vector.	which loop to vectorize array references to vectorize scalars in original input	last_entry(inner_loops)=_5 ref_info[_10]=(RW,reduce), ref_info[_11]=(R,copy,vec), ref_info[_12]=(R,copy,vec) scalar_info[beta]=(R), scalar_info[alpha]=(R)
strength reduce	which array exps to reduce loops enumerating each reference	arr_refs={_10,_11,_12} dims_out[_10]=(_2,_4), dims_out[_11]=(_4,_6), dims_out[_12]=(_2,_6), dims_in[_10]=(_1,_3), dims_in[_11]=(_3,_5), dims_in[_12]=(_1,_5)
loop unrolling	which loops to unroll unroll factor	last_entry(inner_loops)=_5 last_entry(unroll_size)=4
loop splitting	scope to apply cleanup	cleanup_scope=_7
peephole opt.	scope to apply assembly opt.	cleanup_scope=_7

Table I
CUSTOMIZING OPTIMIZATIONS FOR THE KERNEL IN FIG 4(A)

MM_pattern optimizations and the result of using the *CONFIG* variables in Fig 6 to customize each optimizations in Fig 2 for the $gemm$ kernel in Fig 1. Here except for *ref_info*, *scalar_info*, *par_size*, *tile_size*, and *unroll_size*, whose values are not expected to change irrespective of what optimizations have been applied, all the other *CONFIG* variables use coordination handles inserted inside the input code, illustrated in Fig 4(a), as their values. The configurations for all optimizations are determined based on the set of preselected *CONFIG* variables.

C. Programmable Composition of Optimizations

Fig 7 presents our algorithm for composing the 10 optimizations we currently support for *MM_pattern*, after the two functions in Fig 6 have been invoked to collectively customize the preselected optimizations. The algorithm essentially invokes each optimization in Table I one after another in their pre-determined order based on their pre-customized configurations. Each optimization is invoked using the syntax

$$invoke\_opt(p_1, ..., p_m, r_1 = v_1, ..., r_n = v_n),$$

where *opt* is the name of the optimization (e.g., *blocking* or *parallelization*), p_1,...,p_m are required input parameter values (e.g., ℓ_1), r_1,...,r_n are names of optional output or tuning

```
optimize_MM_pattern(pat: MM_pattern code to optimize)
(1) ℓ1=first_entry(CONFIG.outer_loops);
    invoke_blocking(ℓ1,ret_new_vars=CONFIG.private_vars,factor=CONFIG.par_size);
    invoke_parallelization(CONFIG.parallel_loop, CONFIG.private_vars);
    move_handle(ℓ1, loop_body(ℓ1)); move_handle(CONFIG.cleanup_scope, ℓ1);
(2) if (sizeof(CONFIG.outer_loops) > 1) then
      invoke_blocking(CONFIG.outer_loops, ret_new_ivars=CONFIG.private_vars,
                      ret_inner_tile=CONFIG.inner_loops, factors=CONFIG.tile_size);
    endif
(3) foreach entry arr→ exp in CONFIG.arr_copy do
      r=arr_ref_handle(exp); loops=CONFIG.dims_out[r]; info=CONFIG.ref_info[r];
      if copy_triangular(info) then loc=CONFIG.parallel_top; vars=null;
      else loc=ℓ1; vars=CONFIG.private_vars; endif
      invoke_array_copy(exp,loops,loc,has_read(info),has_mod(info),ret_new_vars=vars);
    enddo
(4) if (sizeof(CONFIG.inner_loops) > 1) then
      invoke_unroll&jam(CONFIG.inner_loops, factor=CONFIG.unroll_size);
    endif
(5) foreach array reference r ∈ CONFIG.arr_refs do
      loops=CONFIG.dims_in[r]; info=CONFIG.ref_info[r];
      loc=loop_body(last_entry(loops)); vars=∅;
      invoke_scalar_repl(r,loops,loc,has_read(info),has_mod(info),ret_new_vars=vars);
      CONFIG.private_vars∪ = vars; CONFIG.scalar_info[vars] = info
    enddo
(6) if  vectorize(info) == true ∀ vars → info ∈ CONFIG.scalar_info then
      scope = loop_body(second_to_last_entry(CONFIG.inner_loops));
      invoke_simd(CONFIG.scalar_info, scope, ret_new_vars=CONFIG.private_vars);
    endif
(7) foreach array reference r ∈ CONFIG.arr_refs do
      loops = append(CONFIG.dims_outer[r], CONFIG.dims_in[r]);
      invoke_strength_reduction(r, loops, ret_new_vars=CONFIG.private_vars);
    enddo
(8) invoke_loop_unroll(last_entry(CONFIG.inner_loops), factor=last_entry(unroll_size));
(9) invoke_loop_splitting(CONFIG.cleanup_scope);
(10)invoke_peephole_optimization(CONFIG.cleanup_scope);
   enddo
```

Figure 7. Algorithm: composition of optimizations

parameters (e.g., *ret_new_vars* and *factor*), and v_1,...,v_n are values for the optional output or tuning parameters.

A key emphasis of the algorithm is that after each step, all the coordination handles in the *CONFIG* variables are adjusted if necessary to make sure they always have the correct values. For example, *CONFIG.private_vars* is explicitly modified after each optimization that may create new variables, through the ret_new_vars parameter of loop blocking, array copy, unroll&jam, scalar replacement, and SIMD vectorization. The other handles are maintained in a similar fashion through two steps. First, when each optimization is invoked, it ensures all the coordination handles in the input are modified with equivalent transformed code. Second, after each optimization, the composition algorithm explicitly modifies various coordination handles to synchronize among the optimizations, as detailed in the following.

1) Loop parallelization: the first entry (ℓ_1) of *CONFIG.outer_loops* is extracted, strip-mined using *CONFIG.par_size* as the blocking factor, and then parallelized via OpenMP. Then, the coordination handle in ℓ_1 and the *CONFIG.cleanup_scope* handle are explicitly moved to the inner tiled loops (body of ℓ_1) for additional thread-local optimizations.
2) Loop blocking: if *CONFIG.outer_loops* has more than one loops, they are blocked (using *CONFIG.tile_size* as the blocking factors), and *CONFIG.inner_loops* is passed as an output parameter of *invoke_blocking* so that its handles are moved to the inner tiled loops.

3) Array copy: for each array in *CONFIG.arr_copy*, if only the lower/upper triangular of the array needs to be copied, the copies are performed sequentially before the parallelized loop; otherwise, each thread will make its own local copy of the array before the outermost loop in *CONFIG.outer_loops*. The new layout of the copied array is made to reflect how the blocked loops in *CONFIG.dims_out* enumerate different elements of the array. and *CONFIG.private_vars* is modified to contain new local variables if copying is performed concurrently by each thread.
4) Loop unroll&jam: if *CONFIG.inner_loops* has more than one loops, the outer ones are unrolled (using *CONFIG.unroll_size* as unrolling factors), with the unrolled iterations jammed inside the innermost loop.
5) Scalar replacement: replace each array reference r in *CONFIG.arr_refs* with scalar variables inside the body of its innermost nonzero-projected loop (*last_entry(CONFIG.dims_inner[r])*), according to configurations in *CONFIG.ref_info[r]*. Both *CONFIG.private_vars* and *CONFIG.scalar_info* are then modified to include the new scalar variables created.
6) SIMD vectorization: if all variables in *CONFIG.scalar_info* can be safely vectorized, the innermost loop in *CONFIG.inner_loops* is vectorized, and the body of its surrounding loop is modified to use SSE/AVX registers. Note that all the array references have been replaced with scalars, so their information can be found in *CONFIG.scalar_info*.
7) Strength reduction: for each entry r in *CONFIG.arr_refs*, incrementally evaluate the referenced address at each iteration of the loops in *CONFIG.dims_out[r]* or *CONFIG.dims_in[r]*, whose index variables are used in r to enumerate different elements.
8) Loop unrolling: the last entry in *CONFIG.inner_loops* is unrolled to create a larger loop body.
9) Loop splitting: examine the so-far optimized code in *CONFIG.cleanup_scope* and invoke loop splitting to eliminate conditionals inside loops.
10) Peephole optimization: invoke dynamic assembly level optimizations to further improve efficiency within *CONFIG.cleanup_scope*.

The purpose of the loop splitting and peephole optimizations in Fig 7 is to further improve the efficiency of the code generated by previous optimizations. Unlike the other optimizations, they do not have a set configuration but instead examine their input code to dynamically identify optimization opportunities. In particular, when each of the earlier optimizations generate code fragments that need to be later cleaned up, special symbolic names, e.g., ℓ_1, ℓ_2, and ℓ_3 in Fig 4(b), and relations, e.g., $tile(\ell_1)$, $split\_at\_begin(tile(\ell_3, 1))$, and $unroll\_check(tile(tile(\ell_1)))$, are created to tag these fragments, thereby dynamically driving the operations of the later cleanup optimizations.

```
invoke_loop_splitting(input: code to cleanup)
(1) foreach loop ℓ1 from inside out in input do
      t1 = get_loop_ctrl_tag(ℓ1); body=get_loop_body(ℓ1);
      if (t1 ≠ null) then
(1.1)  (ℓ2,x) = find_last_loop_with_tag(split_at_end, tile(t1),body));
       if ℓ2 ≠ null then peel_loop_at_end&cleanup(ℓ1, x); endif
(1.2)  (ℓ2,x) = find_last_loop_with_tag(unroll_check, tile(t1),body));
       if ℓ2 ≠ null then split_loop_at_tile_end&cleanup(ℓ1, x); endif
(1.3)  (ℓ2,x) = find_last_loop_with_tag(split_at_begin, tile(t1),body));
       if ℓ2 ≠ null then peel_loop_at_begin&cleanup(ℓ1, x); endif
    enddo
(2) foreach loop ℓ1 from inside out in input do
      t1 = get_loop_ctrl_tag(ℓ1); body=get_loop_body(ℓ1);
      if (t1 ≠ null) then
       (ℓ2,x) = find_last_loop_with_tag(unroll_check, t1,body));
       if ℓ2 ≠ null then split_loop_at_end&cleanup(ℓ1, x); endif
      endif
    enddo
```

Figure 8. Algorithm: cleaning up conditionals inside loops

Fig 8 shows an algorithm skeleton of the *invoke_loop_splitting* function invoked in Fig 7, which serves to remove conditionals, e.g., those tagged with the *split_at_begin* and *unroll_check* relations in Fig 4, from inside loops, by splitting the surrounding loops whose index variables are involved in the conditionals, so that the conditionals become either always true, in which case the check can be removed, or always false, in which case the entire conditional together with its body are removed. In particular, step (1) of the algorithm removes all conditionals introduced by loop blocking by identifying each outer blocked loop that has been tagged with a symbolic name, e.g., ℓ_1, with some loop in its body tagged with $tile(\ell_1)$, and its body contains at least one of the relational tags, *split_at_end(tile(ℓ_1))*, *unroll_check(tile(ℓ_1))*, or *split_at_begin(tile(ℓ_1))*. An artificial ordering is set for the splitting process so that each loop is first split at the end if necessary, before the main loop is split again at the beginning. Step (2) of the algorithm removes the *unroll_check* conditionals introduced by loop unrolling or unroll&jam in a similar fashion.

We currently apply only two peephole optimizations. The first aims to eliminate an extra penalty introduced by Intel processors [16] when multiple data items in an AVX register are reduced to a single value, using an SSE instruction. In particular, a *vzeroupper* instruction is inserted after the reduction operation to prevent the hardware from recovering the upper bits of the AVX register, thus saving wasted clock cycles. The second optimization aims to translate each pair of multiply-followed-by-an-add instructions into a single fused-multiply-accumulate instruction (FMA) on AMD processors [3], thereby removing the extra dependences and leading to improved pipeline efficiency.

D. Correctness, Generality And Effectiveness

Our current specialization of compiler optimizations includes two key strategies: (1) a customized ordering of the optimizations and (2) a specialized configuration for each optimization, to collectively maximize their effectiveness while minimizing interferences. In particular, the following

decisions are made to customize a set of general-purpose compiler optimizations for dense matrix computations.

- The outermost loop is first strip-mined before parallelized, so that when the input code does not include sufficient work (i.e., the loop iteration count is low), it does not have to be parallelized using all the threads.
- Array copy is enabled together with loop blocking but only for arrays that carry sufficient reuses.
- When possible, copying of arrays are performed concurrently by multiple threads, to reduce the startup cost of each thread by preloading the copied arrays.
- Blocking and unrolling factors are determined by empirically selecting from a set of best-known values.
- Loop splitting and peephole optimizations are specially designed to cleanup results of earlier optimizations.

All the optimizations we consider are general purpose and are likely already included in a majority of optimizing compilers (e.g., PLUTO [9] + icc). So the benefit of specialization comes only from pattern-based collective customization of these optimizations and the fine-grained coordination among optimizations enabled by our framework. The correctness of the specialization is guaranteed by two conditions: (1), the consistency between our pattern analysis algorithm in Fig 5 and optimization customization algorithm in Fig 6, as the analysis never mis-categorizes an input code as *MM_pattern* unless all the pre-customized optimizations in Fig 6 can be safely applied to the computation, with additional runtime checks inserted surrounding the optimized code to resolve any remaining uncertainty; and (2), the correctness of our optimization composition algorithm in Fig 7, which uses fine-grained coordination among the optimizations to guarantee that the original optimization customizations remain correct irrespective of any intermediate modifications to the input code. The following further details the correct customization of each specialized optimization.

- *OpenMP parallelization*, which is applied to the outermost loop of the original input. Since our *MM_pattern* requires that this loop cannot carry any dependence, and lines 7 and 31 of Fig 6 modify *CONFIG.private_vars* to contain all loop index variables and private variables within each iteration, the optimization is safe.
- *Loop blocking and unroll&jam*, which are applied if the input code contains more than one loop. Their correctness requires all the loops to be nested inside one another, and each loop can only carry forward dependences from earlier iterations to later ones. Both conditions are guaranteed by our pattern analysis.
- *Array copy and scalar replacement*, which are correct as long as each array reference can be precisely mapped to its new locations, and the new locations are properly initialized and saved back to the original arrays when necessary. The precise location mappings are guaranteed by our pattern analysis of the input code, which requires that data accessed by each array reference is an orthogonal projection of its surrounding loops. The *CONFIG.ref_info* variable contains the result of invoking the *categorize_uses(r)* function at line 25 of Fig 6 to determine correct copy configurations for each array reference r based on its use patterns.
- *SIMD vectorization*, which parallelizes the innermost loop if all the array references are categorized as vectorizable by *CONFIG.ref_info*; i.e., they either access the same element or contiguous memory locations at consecutive iterations of the loop. Since our pattern analysis guarantees the innermost loop does not carry any non-reduction dependence, the optimization is safe.
- *Strength reduction, innermost loop unrolling, loop splitting, and peephole optimizations*, which are either always correct (e.g., loop unrolling) or use internal symbolic analysis of expressions to guarantee correctness of the optimizations.

	Level-1	Level-2	Level-3
Applicable	axpy, scal, copy, swap	gemv, ger, trmv, trsv	gemm, syrk, syr2k, trsm(l/u), trmm(l/u)
+ Parallel reduction	asum, dot, min, max, norm, abs		
+ Data-layout normoalization		symv, sbmv, spmv, syr, syr2, spr, spr2 gbmv, tbmv, tbsv, tpmv, tpsv	symm

Table II
CATEGORIZATION OF BLAS ROUTINES

Our definition of the *MM_pattern* is based on a set of constraints on the input code that are required to trigger a special customization of optimizations well recognized as important for linear algebra kernels. The same methodology can be similarly applied to specialize the optimization of other patterns of computation, e.g., stencils and digital signal processing, and graph algorithms, by recognizing how to best optimize these computations, formulating the constraints explicitly, and then replacing our algorithms in Figures 5, 6, and 7 accordingly.

The *MM_pattern* we currently target applies to a large number of fundamental linear algebra kernels. In particular, Table II shows the categorization of the entire collection of BLAS routines into the following three classes.

1) Applicable: these kernels belong to our *MM_pattern* and include 40% of level-1, 25% of level-2, and 85% of level-3 BLAS kernels; overall 44% of all BLAS.
2) + Parallel reduction: these kernels cannot be categorized as *MM_pattern* because their outermost loops carry cross-iteration dependences and thus cannot be parallelized in the same straightforward fashion.
3) + Data-layout normalization: these kernels cannot be categorized as *MM_pattern* because their input matrices use packed, banded, or symmetric storage format. As the result, the array subscript expressions used to reference these matrices are no longer orthogonal

projections of the surrounding loop index variables.

While the BLAS library includes only a subset of all dense matrix computations, it provides the most fundamental building blocks that are combined to solve more complex problems, e.g., LU/QR factorizations and eigenvalue solvers in LAPACK [5]. Our framework, as all general-purpose compilers, aims to automatically recognize and optimize all occurrence of such kernels in higher-level routines and large applications. Our current optimizations target only commodity multi-core Intel/AMD CPU processors. However, the optimization customization and composition algorithms in Figs 6 and 7 can be extended to alternatively target new emerging architectures such as many-core GPUs.

IV. Experimental Results

We have implemented a prototype of our optimization framework using POET [32], a program transformation language designed to support the programmable control of compiler optimizations. Our optimizer invokes the POET built-in library to perform a standard set of loop optimizations [2] and the SIMD vectorization algorithm in [15]. Our algorithms for pattern analysis (Fig 5), collective customization(Fig 6), and optimization composition (Fig 7) are all implemented from scratch using POET. The blocking and unrolling factors are automatically determined using the transformation-aware empirical-search algorithm in [25] based on the performance feedback of differently optimized code. Specifically, different values of these parameters may be chosen for each loop being optimized by our framework to maximize performance of the optimized code.

CPU	Freq.	L1 cache sz	L2 cache sz	# of cores
Intel Xeon E5-2420	1900MHz	32KB	256KB	12
AMD FX(tm)-8320	1400MHz	16KB	2048KB	8

Table III
Platforms configuerations

	Intel Xeon E5-2420	AMD FX(tm)-8320
# threads	12	8
input sz	7680^2 (level-3)	5120^2 (level-3)
	10240^2 (level-2)	10240^2 (level-2)
	1024000 (level-1)	1024000 (level-1)
gcc-4.7.3 flags	-O2 -fopenmp	-O2 -fopenmp
icc-14.0.0 flags	-O3 -ftree-vectorize -openmp	-

Table IV
Evaluation configurations on both Intel and AMD

We have used our framework to optimize all the 15 BLAS kernels that belong to *MM_pattern*, listed under the *Applicable* category in Table II, and 9 application benchmarks from the SPLASH-2 (Stanford ParalleL Applications for SHared memory) benchmark suite [30]. Differently optimized implementations of the kernels and applications are evaluated on two machines, an Intel E5-2420 and an AMD FX(tm)-8320, shown in Table III. The execution configurations for all the kernels are shown in Table IV, and those for the 9 application benchmarks are shown in Table V. Each implementation has been evaluated five times, with the average performance reported (the variation across runs is 2-8%).

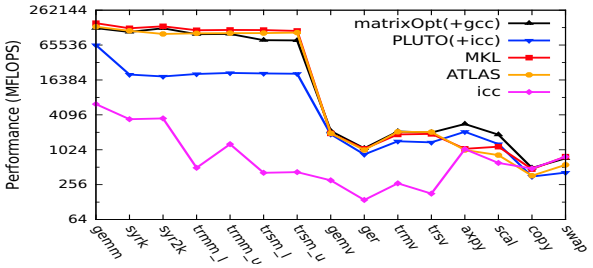
For each BLAS kernel, we compare the performance of our automatically optimized kernels with that of the manually optimized ones from three leading BLAS library implementations, Intel MKL-11.0 [17], AMD ACML-5.3.1 [4], and ATLAS-3.11.11 [29], and with that automatically attained by two compilers, PLUTO-0.7 [9] and *Intel icc-14.0.0*. Then, the impact of the optimization specialization and the scalability of the optimized kernels are studied in more details. For each SPLASH-2 benchmark, we compare the optimization speedups attained by our framework with those attained by PLUTO and icc.

	Intel Xeon E5-2420	AMD FX(tm)-8320
#threads	12	8
compiler	icc (-O3 -lpthread -lm)	gcc (-O3 -lpthread -lm)
Barnes	64K particles	
Fmm	64K particles	
Ocean(cont/ncont)	1026×1026 ocean	
Raytrace	car	
Volrend	head	
Water(nsq/sp)	4096×4096 molecules	
Radiosity	room, -ae 5000.0 -en 0.050 -bf 0.10	

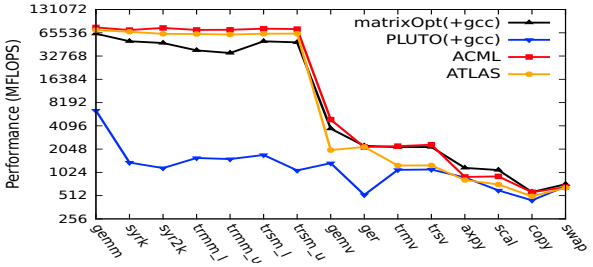
Table V
Evaluation configurations of SPLASH-2 applications

A. Overall Performance

Fig 9 compares the overall performance of the 15 BLAS kernels when optimized by our *matrixOpt* optimizer, the PLUTO [9] compiler, and the Intel icc compiler (used only on the Intel platform). The compiler optimized kernels are also compared with manual implementations from the Intel MKL, AMD ACML, and ATLAS libraries, The MFLOPS of the benchmarks are plotted using logarithmic base-2 scale to highlight the performance differences at all levels.



(a) Intel Xeon E5-2420



(b) AMD FX(tm)-8320

Figure 9. Overall performance of optimized kernels

For the 7 level-3 BLAS kernels, our optimizer attained 68-90% of the best performance attained by the MKL library on the Intel Xeon, 63-84% of the best performance by the ACML library on the AMD, and 73-94% of the performance of ATLAS on both processors. The performance automatically attained by our optimizer is significantly better than those attained by the PLUTO and the Intel icc compilers, by factors of 2-40 and 20-208 respectively. The difference between our optimized code and those optimized through manually written assembly in MKL, ACML, and ATLAS is due to the lack of some additional dynamic optimizations, e.g., memory prefetching and instruction scheduling, that our optimizer currently does not support.

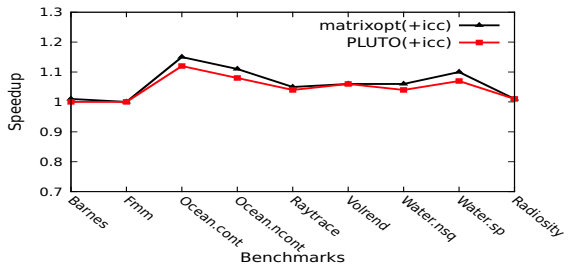
For the 8 level-2 and level-1 BLAS kernels, the performance attained by our optimizer is the best among all implementations on both platforms except for *gemv_n* on the AMD, where our optimizer attained 75% of the best performance attained by ACML. The performance by PLUTO is only slightly worse than that of our matrixOpt on the Intel processor but lags significantly behind the other implementations on the AMD, due to the lack of backend optimization support by gcc. In contrast, our optimizer performs equally well irrespective of whether gcc/icc is used to generate machine code, as it collectively integrates a much wider spectrum of optimizations in a coordinated fashion.

To quantify the benefit of integrating pattern-based optimization specialization within a general-purpose compiler to optimize large applications, Figure 10 shows the extra speedups gained by optimizing the 9 SPLASH-2 benchmarks using our matrixOpt or PLUTO combined with icc/gcc compilers, over using icc/gcc alone. Our matrixOpt has successfully identified *MM_patterns* in 5 of the 9 benchmarks, where it attained 5%-15% extra speedup over using icc alone on the Intel processor and 4%-11% extra speedup over using gcc along on the AMD. Although the loops optimized by our matrixOpt are a subset of those optimized by PLUTO, our optimized benchmarks attained an extra 2-4% speedup over those by PLUTO on both processors.

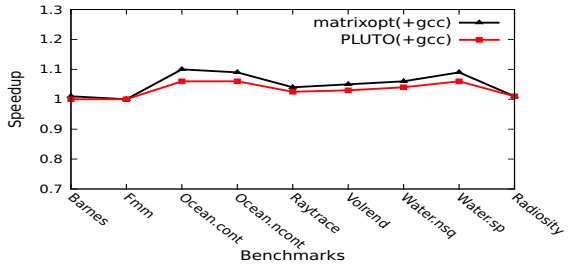
B. Impact Of Individual Optimizations

To better explain the performance differences in Fig 9, Fig 11 quantifies the contribution of each individual matrixOpt optimization when applied to the BLAS kernels on the Intel platform, by turning off each optimization one by one in the reverse of their application order in Fig 7, until only OpenMP parallelization is left. The impact of optimizations is similar on the AMD and is omitted here.

For the 7 level-3 BLAS kernels, the 3 CPU-level optimizations, *SIMD vectorization*, *Loop splitting* and *Peephole Optimization*, have the most impact as these kernels have become CPU-bound after the earlier optimizations. The PLUTO compiler applied the loop optimizations well to enhance cache reuse. However, since it relies on Intel icc to perform the backend CPU-level optimizations, the overall



(a) Intel Xeon E5-2420



(b) AMD FX(tm)-8320

Figure 10. Effectiveness of optimizing SPLASH-2 benchmarks

performance is compromised due to the lack of coordination among the two compilers. The icc compiler performs many advanced CPU-level optimizations. However, using icc alone attains the worst performance among all approaches due to the lack of better cache-level optimizations.

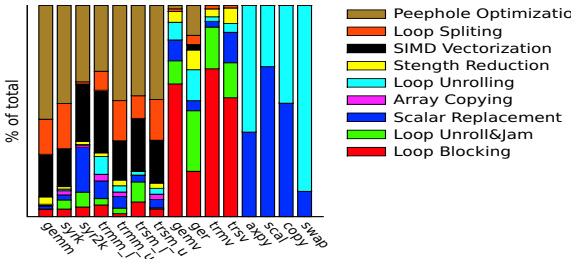


Figure 11. Contribution of optimizations on Intel Xeon

For the 4 level-2 BLAS kernels, the most important optimizations are loop blocking and unroll&jam, which enhance data reuses within the computation. Because these kernels remain memory-bound even after blocking, the CPU-level optimizations are useful but do not play nearly as important a role as that for the level-3 kernels. Here reusing each data item is critical to attaining performance, a job all optimization approaches did reasonably well in Fig 9.

For the 4 level-1 BLAS kernels, only a single loop is evaluated in the computation. Since they are severely memory-bound and without any reuse of data, neither the cache level nor the CPU-level optimizations have any impact, and the most important ones are scalar replacement and loop unrolling. Our optimizer attained better performance for these kernels likely because it did a good job of replacing array references with scalars.

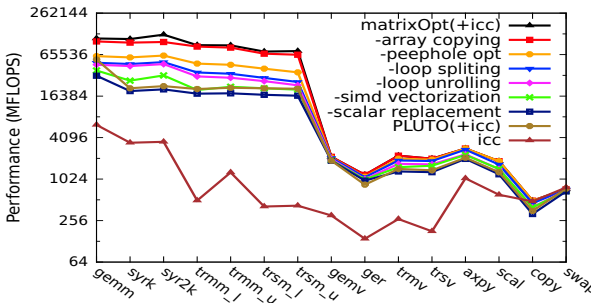


Figure 12. Impact of optimization coordination on the Intel Xeon

C. Impact Of Optimization Coordination

If taking out array-copy, the optimizations supported by our matrixOpt are a strict subset of those supported by PLUTO + icc. All combined together, they are a strict subset of those manually applied by MKL, ACML, and ATLAS. Therefore, our performance speedups over PLUTO+icc can only come from two sources: (1) the benefit of applying array copy together with the other optimizations; and (2) the fine-grained coordination among all optimizations. Fig 12 aims to quantify the impact of both sources by combining our matrixOpt with the Intel icc compiler instead of gcc. Then, after removing the array copy optimization from our matrixOpt, we incrementally remove each optimization already implemented in icc to quantify the impact of invoking the optimization with or without coordination.

From Fig 12, after disabling the array copy optimization, we observed 15%-29% slowdown for the level-3 kernels. The slowed down matrixOpt, however, still outperformed PLUTO+icc by factors of 1.6-31, which quantify the benefit of fine-grained coordination and specialized customization of the optimizations. Additionally disabling the coordinated application of 3 CPU-level optimizations, peephole, loop splitting, and SIMD vectorization, however, significantly slowed down the optimized code. Finally, after further disabling scalar replacement, where the optimizations in matrixOpt become a subset of those applied by PLUTO, our stripped-down matrixOpt performed worse than PLUTO+icc by 3%-40% for the level-3 BLAS kernels.

D. Impact Of Optimization Ordering

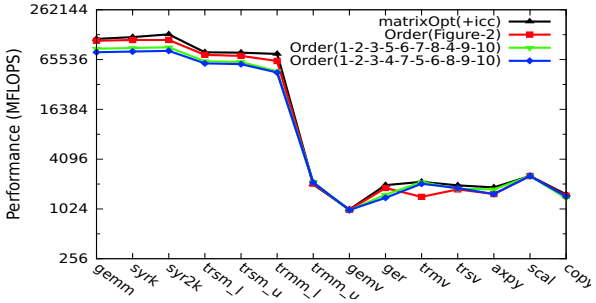


Figure 13. Impact of optimization ordering on the Intel Xeon (the indexing of the involved optimizations is based on those in Fig 7)

Our matrixOpt applies 10 finely-coordinated optimizations in a pre-customized order, shown in Fig 7. Fig 13 quantifies the impact of this specialized ordering of the coordinated optimizations by comparing its performance with that of three alterantive orderings of the same optimizations, the ordering in Fig 2, which represents a natural ordering of the involved optimizations in a general-purpose compiler, and two variations of the default ordering in matrixOpt: the first variation moves loop unroll&jam (optimization#4) to be applied after scalar replacement (#5), SIMD vectorization (#6), strength reduction (#7), and loop unrolling(#8); the second variation moves strength reduction (#7) to be applied before scalar replacement (#5) and SIMD vectorization (#6).

From Fig 13, the specialized optimization ordering in matrixOpt is able to attain 3%-20% better performance than that of the ordering in Fig 2. Further, two small variations of the best ordering incurred 4%-45% and 6%-67% slowdown respectively, indicating the sensitivity of the matter. In particular, even when the optimizations are able to adequently coordinate with each other, the ordering of a few optimizations, specifically, loop unroll&jam, scalar replacement, and SIMD vectorization, can result in significant differences in the number of scalar variables, and thereby the register pressure, of the loop body, which in turn can result in different registers being spilled and different numbers of pipeline stalls due to instruction scheduling,

E. Scalability Of Optimized Code

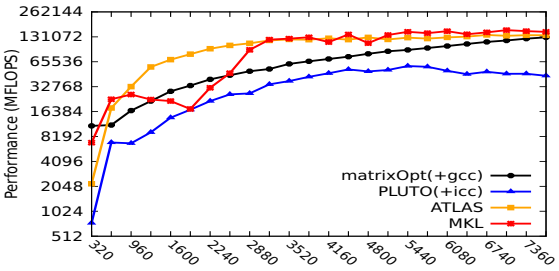
To study the weak scalability of the optimized kernels, Fig 14 shows the performance in logarithmic base-2 scale of two kernels, $gemm$ and $gemv$, which represent level-3 and level-2/1 BLAS groups, on the Intel Xeon using matrix sizes ranging from 320^2-7680^2 and 512^2-10240^2 respectively.

For $gemm$, our optimized code performed worse than that of MKL/ATLAS but has continued to improve as the matrix size becomes increasingly large. This is likely due to our array copy optimization, which incurs a higher overhead than those employed by MKL and ATLAS. However, since each copied item is reused N times, the overhead is increasingly amortized as the matrices become larger.

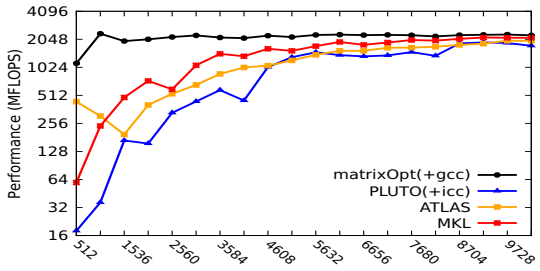
For $gemv$, our performance stays mostly constant for all matrix sizes, while the other implementations performed poorly for small matrices but steadily increased their performance as the size becomes larger. Here because our optimization algorithm in Fig 7 explicitly blocks the outermost loop into chunks of iterations before invoking OpenMP parallelization, it guarantees the amount of work by each thread and elects to use fewer threads when insufficient amount of computation is available. As the result, its performance does not vary with the problem size as the other approaches do.

V. Related Work

Many compiler optimizations, e.g., loop blocking [20], unroll&jam [10], scalar replacement [27], SIMD vectoriza-



(a) gemm



(b) gemv

Figure 14. Scalability of optimized code on the Intel Xeon

tion [15], [26], [23], among others [22], [12], have been developed for dense matrix codes. Our work integrated these optimizations and aims to enhance their aggregated effectiveness through collective customization and coordinated composition of the optimizations.

A number of general frameworks[18], [1], [21], e.g., the polyhedral model [7], has been shown to be effective at collectively considering a large set of the loop optimizations and generating efficient code. However, this body of work is limited in that they consider only loop optimizations. As shown in our experimental results, the performance attained by the PLUTO Polyhedral optimizing compiler [9] lags significantly behind those attained by our specialized optimizer and HPC libraries due to its lack of fine-grained coordination with the later backend optimizations by the icc compiler. Our work compliments these frameworks by providing additional mechanisms to enable such fine-grained coordinations across all optimizations.

We implemented our prototype optimizer using the POET program transformation language [32]. Yi and Whaley demonstrated that by manually writing POET scripts to optimize several linear algebra kernels, they can achieve performance comparable to that achieved by manually written assembly in ATLAS [33]. Yi [31] developed a source-to-source optimizing compiler to *automatically* produce parameterized POET scripts for a subset of the optimizations. Similarly, the EPOD framework by Cui et. al [14] used two patterns, a dense matrix and a stencil pattern, to specialize optimizations within the Open64 compiler. Our work also seeks pattern-guided specialization of compiler optimizations. We consider a larger set of optimizations and a wider variety of kernels than those considered previously for dense matrix computations. Further, we automatically perform collective customization and specialization of the optimizations, which were not done previously. Qian et. al developed AUGEM [28], a framework that uses a template-based approach to automatically integrate specially tailored assembly-level optimizations usually applied only manually by HPC developers. Their framework also uses the POET language but focuses on domain-specific code generation instead of general-purpose compiler optimizations.

This paper is orthogonal to existing research on automated empirical tuning of compiler optimizations through *iterative compilation* [6], [24], [11]. Our optimizer currently uses a simple search script [25] to empirically determine appropriate blocking and unrolling factors but can easily integrate other existing more advanced empirical tuning techniques.

VI. Conclusions

This paper presents a new methodology to support the collective customization and fine-grained coordination across the spectrum of general-purpose compiler optimizations and demonstrates the effectiveness of this methodology by collectively customizing and specializing 10 compiler optimizations to attain a highest level of performance for dense matrix computations. Our future work will integrate additional optimizations to target computations in other domains, e.g., stencils and sparse matrices, and for additional computing platforms such as many core GPUs.

Acknowledgment

This research is funded by the USA National Science of Foundation under grants CCF-1261778, CCF-1261811, and CCF-1421443, and by the USA Department of Energy under grant DE-SC0001770.

References

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, Santa Fe, New Mexico, May 2000.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, Oct 2001.

[3] AMD Corporation. *New Bulldozer and Piledriver Instructions*, 2012.

[4] AMD Corporation. *AMD Core Math Library (ACML)*, 2013.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. The Society for Industrial and Applied Mathematics, 1999.

[6] N. Baradaran, J. Chame, C. Chen, P. Diniz, M. Hall, Y.-J. Lee, B. Liu, and R. Lucas. Eco: An empirical-based compilation and optimization system. In *International Parallel and Distributed Processing Symposium*, 2003.

[7] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'04:Parallel Architecture and Compilation Techniques*, pages 7–16, 2004.

[8] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 59:1–59:12. ACM, 2009.

[9] U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, 2008.

[10] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[11] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*, March 2005.

[12] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 151–160, New York, NY, USA, 2005. ACM.

[13] K. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[14] H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan. Extendable pattern-oriented optimization directives. In *CGO*, 2011.

[15] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 82–93, New York, NY, USA, 2004. ACM.

[16] Intel Corporation. *Avoiding AVX-SSE Transition Penalites*, 2011.

[17] Intel Corporation. *Intel Math Kernel Library Reference Manual*, 2012.

[18] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997.

[19] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, LCTES '03, pages 12–23, New York, NY, USA, 2003. ACM.

[20] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, Apr. 1991.

[21] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[22] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[23] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.

[24] Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *Proc. Parallel Architectures and Compilation Techniques*, 2006.

[25] S. F. Rahman, J. Guo, and Q. Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *HIPEAC'11: High-Performance and Embedded Architectures and Compilers*, Heraklion, Greece, Jan 2011.

[26] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 165–175, Washington, DC, USA, 2005. IEEE Computer Society.

[27] K. K. Steve Carr. Scalar replacement in the presence of conditional control flow. *Software — Practice and Experience*, 24(1):51–77, Jan. 1994.

[28] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 25:1–25:12, New York, NY, USA, 2013.

[29] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.

[31] Q. Yi. Automated programmable control and parameterization of compiler optimizations. In *CGO'11: ACM/IEEE International Symposium on Code Generation and Optimization*, Apr. 2011.

[32] Q. Yi. POET: A scripting language for applying parameterized source-to-source program transformations. *Software: Practice & Experience*, pages 675–706, May 2012.

[33] Q. Yi and C. Whaley. Automated transformation for performance-critical kernels. In *LCSD'07: ACM SIGPLAN Symposium on Library-Centric Software Design*, Montreal, Canada, Oct. 2007.